Implementation of an Interior Point LP Algorithm on a Shared-Memory Vector Multiprocessor

Matthew J. Saltzman

CRPC-TR91199
November, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Implementation of an Interior Point LP Algorithm on a Shared-Memory Vector Multiprocessor

Matthew J. Saltzman
Department of Mathematical Sciences, Clemson University
Clemson SC 29634-1907, USA
Telephone: 803/656-3434 E-mail: mjs@clemson.edu

Keywords: Linear programming; interior point algorithms; Cholesky factorization; sparse matrices; parallel matrix factorization; linear algebra.

Abstract

One attractive aspect of interior point algorithms for linear programming is their suitability for implementation on multiprocessing computers. This paper describes a number of issues relating to the implementation of these methods on shared-memory vector multiprocessors. Of particular concern in any interior point algorithm is the factorization of a sparse, symmetric, positive definite matrix. This implementation exploits the special structure of such matrices to enhance vector and parallel performance. In initial computational tests, a speedup of up to two was achieved on three processors.

1 Introduction

Since the introduction in the mid-1980s of interior point algorithms for linear programming, significant advances in both simplex and interior point algorithms have dramatically increased the size of problems that can be solved in reasonable time. These advances have occurred in both algorithm design and in exploitation of advanced features of current supercomputers, such as vector processing, for example, see [2, 14, 15, 13], and others. Interior point methods now promise to outperform the simplex method on large instances of certain classes of problems, and on many problems previously considered particularly difficult.

One feature of interior point methods is that they appear to be more amenable than the simplex method to implementation on parallel computers. In this paper, we investigate opportunities for exploiting advanced-architecture computers in the implementation of interior point algorithms. Such opportunities are concentrated in the construction, factorization and solution of a sparse, positive definite (SPD) system of equations. This is the most computationally intensive part of each iteration: in large or dense problems, 80–90% or more of the computation time is spent in this step (on serial machines). This paper describes implementations of vector and parallel algorithms for factoring large, sparse, SPD matrices, and shows how these methods can be integrated into interior point LP algorithms. We describe the results of computational tests, and indicate where further research is likely to show promise.

We begin by briefly reviewing a primal-dual interior point LP algorithm. The following section reviews algorithms for solving sparse, SPD linear systems. We discuss the performance characteristics of various forms of the Cholesky factorization algorithm for SPD matrices and their suitability for sparse matrices and vector processing. Next we describe the data structures to support the implementation of the solver and exploitation of vector and parallel processing. Finally, we describe our experience implementing the algorithm on an IBM 3090 vector multiprocessor.

2 A Primal-Dual LP Algorithm

The implementation described here is an extension to the FORTRAN-77 interior point software package, OB1[1] (Optimization with Barriers 1). The

[1] OB1 is a trademark of XMP, Inc.

particular form of interior point algorithm implemented in OB1 is the *primal-dual barrier algorithm*, described in [13]. In this section, we briefly recap this algorithm.

The LP to be solved is:

$$\min\{c^T x : Ax = b, 0 \leq x \leq u\}, \tag{1}$$

where A is $m \times n$. We assume for the purpose of exposition that all of the upper bounds u are finite, although this assumption is not restrictive. The inequalities $x \leq u$ in (1) can be replaced by $x + s = u$, $s \geq 0$. The inequalities $x \geq 0$ and $s \geq 0$ can then be eliminated by incorporating them into a logarithmic barrier function. The equality constraints can be relaxed, and their residual vectors incorporated into the objective with Lagrange multipliers. The Lagrangian barrier function is:

$$L(x, s, y, w \mid \mu) = c^T x - \mu \sum_{j=1}^{n} \ln x_j - \mu \sum_{j=1}^{n} \ln s_j - y^T(Ax - b) - w^T(x + s - u). \tag{2}$$

Thus, L represents a family of functions parameterized by μ. L has a zero-valued derivative when

$$\begin{aligned} Ax &= b, & (3)\\ x + s &= u, & (4)\\ A^T y - w + z &= c, & (5)\\ XZe &= \mu e, & (6)\\ SWe &= \mu e, & (7) \end{aligned}$$

where X, S, Z and W are diagonal matrices with entries x_j, s_j, z_j and w_j, respectively, and z is a vector of dual slack variables (defined by (5)). It is apparent that as $\mu \to 0$, (6) and (7) approach the complementary slackness conditions for a pair of optimal solutions to the primal and dual LPs.

Given $x > 0$, $s > 0$, $w > 0$, $z > 0$ and y, and for a fixed value of μ, we can apply a damped Newton method to (3)–(7). The Newton steps for each of the variables then satisfy:

$$\begin{aligned} Ad_x &= b - Ax, & (8)\\ d_x + d_s &= 0, & (9)\\ A^T d_y + d_z - d_w &= c - A^T y - z + w, & (10)\\ Zd_x + Xd_z &= \mu e - XZe, & (11)\\ Wd_s + Sd_w &= \mu e - SWe. & (12) \end{aligned}$$

An iteration of the algorithm consists of solving (8)–(12) in the following steps:

$$\begin{aligned} d_y &= (A\Theta A^T)^{-1}(A\Theta(\rho(\mu) - d_D) + d_P), \\ d_x &= \Theta(A^T d_y - \rho(\mu) + d_D), \\ d_z &= \mu X^{-1}e - z - X^{-1}Z d_x, \\ d_z &= \mu S^{-1}e - w - S^{-1}W d_x, \\ d_s &= -d_x, \end{aligned}$$

(where $\Theta = (S^{-1}W + X^{-1}Z)^{-1}$ and $\rho(\mu) = \mu(S^{-1} - X^{-1})e - w + z$), performing the ratio tests

$$\begin{aligned} \alpha_x &= \min_j\{-x_j/(d_x)_j : (d_x)_j < 0\}, \\ \alpha_s &= \min_j\{-s_j/(d_s)_j : (d_s)_j < 0\}, \\ \alpha_z &= \min_j\{-z_j/(d_z)_j : (d_z)_j < 0\}, \\ \alpha_w &= \min_j\{-w_j/(d_w)_j : (d_w)_j < 0\}, \end{aligned}$$

and updating

$$\begin{aligned} x &:= x + \alpha_P d_x, \\ s &:= s + \alpha_P d_s, \\ y &:= y + \alpha_D d_y, \\ z &:= z + \alpha_D d_z, \\ w &:= w + \alpha_D d_w, \end{aligned}$$

where

$$\begin{aligned} \alpha_P &= 0.995 \min\{\alpha_x, \alpha_s\}, \\ \alpha_D &= 0.995 \min\{\alpha_z, \alpha_w\} \end{aligned}$$

(the factor 0.995 prevents contact with the boundary). At each iteration, the barrier parameter is computed as

$$\mu = \frac{c^T x - b^T y + u^T w}{\phi(n)}$$

where

$$\phi(n) = \begin{cases} n^2 & \text{if } n \leq 5000 \\ n\sqrt{n} & \text{if } n > 5000 \end{cases} .$$

This results in a substantial reduction in μ at each iteration, and the sequence of solutions converges to a primal and dual optimum. The algorithm terminates when the duality gap is sufficiently small, namely

$$\frac{c^T x - b^T y + u^T w}{1 + |b^T y - u^T w|} < 10^{-8}$$

For details of the algorithm (including selection of the initial solution and initial μ) see [13].

3 The Cholesky Decomposition

The most computationally intensive step in each iteration of the primal-dual algorithm (indeed, of any of the interior point algorithms) is the solution of the system $d_y := (A\Theta A^T)^{-1} r$ (where the form of r and the positive diagonal matrix Θ depends on the particular algorithm). The matrix $A\Theta A^T$ is $m \times m$, symmetric and positive definite. In addition, for most realistic LP problems, both A and $A\Theta A^T$ are sparse. Efficient implementation of the primal-dual algorithm depends critically on efficient solution of this system.

Linear systems of the form $x = M^{-1}b$ are not usually solved by forming the explicit inverse of M. This is particularly true if M is sparse, since taking the inverse does not, in general, preserve sparsity. Instead, the system $Mx = b$ is solved directly, by decomposing M into lower- and upper-triangular factors L and U, such that $M = LU$. Then the triangular systems $Lx' = b$ and $Ux = x'$ can be solved efficiently by forward and backward substitution, respectively. If M is symmetric and positive definite, then M can be factored uniquely into symmetric triangular components $\hat{L}$ and $\hat{L}^T$. These matrices are the *Cholesky factors* of M. A slight variation on this theme computes $M = LDL^T$, where $l_{jj} = 1$ for $j = 1, \ldots, m$, D is a positive diagonal matrix and $\hat{L} = LD^{1/2}$. This version has the advantage of not requiring the computation of square roots on the diagonal of $\hat{L}$, and it is the method that we use in our implementation.

3.1 Algorithms for Cholesky Decomposition

The formula for computing each element of L is given by

$$l_{ij} = (m_{ij} - \sum_{k=1}^{j-1} d_{kk} l_{jk} l_{ik})/d_{jj}. \tag{13}$$

Each diagonal element of D is given by

$$d_{jj} = m_{jj} - \sum_{k=1}^{j-1} d_{kk} l_{jk}^2. \tag{14}$$

Note that in order to compute l_{ij} the values of d_{kk}, l_{ik} and l_{jk} for $k < j$ and d_{jj} must already be known. Computing d_{jj} requires l_{jk} and d_{kk} for $k < j$.

As described in [7] the six permutations of the indices i, j and k in (13) and (14) naturally yield three different algorithms for performing Cholesky decomposition, and determine whether each algorithm applies to L stored in row- or column-major order. In *row Cholesky* (ijk and ikj), rows of L are computed successively; each element in the row is computed using the previously-computed rows and previously-computed elements in the same row. The *column Cholesky* algorithm (jik and jki) computes the columns in succession, using previously-computed columns. *Submatrix Cholesky* (kij and kji) uses each column as it is computed to update all subsequent columns. Consideration of workload and memory access patterns suggest that the jki column Cholesky algorithm and a version of the kji form known as the *multifrontal* algorithm [5] are the best candidates for implementation in parallel. A general discussion of the merits of the various forms of the algorithm is contained in [16]. In this paper, we concentrate on the column Cholesky method.

Sparse vector operations require indirect array references. Even on vector processors with hardware for indirect references, these operations are slower than dense (direct reference) operations [4]. The column and submatrix Cholesky algorithms can be modified to take advantage of the structure of L to replace some sparse SAXPYs with dense SAXPYs (see Section 4). In addition, the multifrontal method replaces all sparse SAXPYs with dense operations. The algorithm performs partial factorizations of a sequence of small, dense *frontal matrices*. The cost to obtain this advantage is a requirement for additional memory to store intermediate, partially-factored submatrices, and the need to assemble the intermediate results to form new

frontal matrices. In this paper, we will be concerned only with the column Cholesky algorithm.

An important aspect of sparse matrix decomposition is the maintenance of sparsity in the resulting factors. In the case of Cholesky decomposition, the positions of nonzeros in the lower triangle of M are a subset of the positions of nonzeros in L. *Fill-in* (non-zeros in L corresponding to zero elements of M) is static, essentially independent of the values of the nonzeros in M, and highly dependent on the ordering of the rows and columns in M. If $M = AA^T$ then the permutation of rows and columns of M is determined by the permutation of rows of A. The Cholesky decomposition routines used up to now in implementations of interior point LP algorithms, *e.g.*, [1, 2, 14], have used graph-based heuristics such as *minimum degree* or *minimum local fill-in* [8] to order the rows of A so as to minimize fill-in. OB1 implements the *multiple minimum-degree* ordering heuristic of [10]. This heuristic is much faster than *minimum degree*, but gives results of comparable quality. The remaining discussion assumes that the row/column permutation of M, and hence the pattern of nonzeros in L, is fixed in advance.

4 Implementation

In this section, we describe the special structure of the Cholesky factor and techniques for exploiting this special structure to improve parallel and vector performance of the column Cholesky algorithm.

4.1 Sparse SAXPY

The key operation in almost all of the linear algebra in the primal-dual algorithm is SAXPY, an operation of the form $y = \alpha x + y$, where x and y are vectors and α is a scalar. If x and y are dense vectors, this operation is a natural one for vectorization. If x and y are sparse, each is stored as a list of indices of nonzero entries and a list of the corresponding nonzero values. The SAXPY operation is performed by copying the entries of y into their positions in a dense work array, then multiplying the entries in x by α and summing them into the corresponding positions in the work array. Finally, the result is copied back to the sparse data structure for y. In the column and submatrix Cholesky algorithms, the nonzero elements of x are a subset of the nonzero elements of y, so the updated y contains no new nonzeros and can be returned to its original location in memory.

If the work array corresponds to the computer's vector registers, the copy operation from memory (the sparse data structure) to the registers is referred to as a *gather*, and the copy from the registers back to memory is called a *scatter*. Most new vector computers implement some form of gather/scatter operations in hardware, but some early vector computers (such as the Cray 1) did not. (The IBM 3090 instruction set provides indirect vector load and store operations.) Even so, indirect operations are slower than direct ones, because an indirect reference must be performed to find a nonzero location in the work array, and because relatively few elements of the vector register may be involved in computing the new result. Vectorization of the sparse SAXPY is still a dramatic improvement over scalar processing (as we will show; see also [9]), but it is still inefficient compared to the dense version of the operation.

4.2 The Elimination Tree

The *elimination tree* T_M associated with the matrix $M = A\Theta A^T$ is a rooted tree constructed from the following relation among the columns of the Cholesky factor L of M [12]:

$$Parent(j) = \min\{i \mid l_{ij} \neq 0, i > j\}$$

Column m of L is taken as the root. For example, consider the matrix depicted in Figure 1. The subdiagonal elements of M are denoted by "•" and the fill-in elements generated during the factorization are denoted by "o". The elimination tree T_M of this matrix is pictured in Figure 2. Columns of L and their corresponding nodes in T_M are referred to interchangeably in the following discussion.

The elimination tree represents a partial ordering of the columns of L, indicating the requirement that the children of column j be computed before column j can be computed itself. This structure will be used to schedule columns to be computed in parallel, as described below. In addition, the elimination tree can be used to detect the presence of supernodes, as described in the next section.

4.3 Supernodes

In order to reduce the number of sparse vector operations, we can exploit the occurrence in L of consecutive columns with the same pattern of nonzero entries. Such groups of columns are called *supernodes* (columns that don't

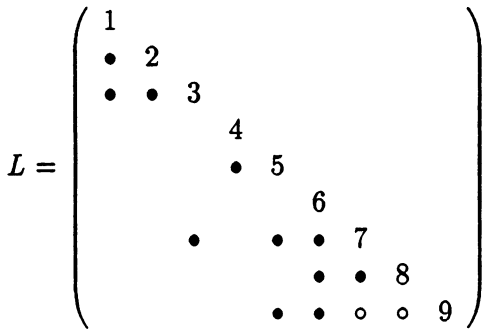

Figure 1: Nonzero structure of a matrix M and its Cholesky factor L.

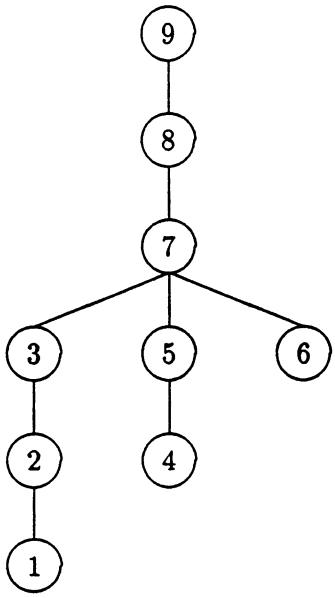

Figure 2: Elimination tree $T(M)$ of the matrix in Figure 1.

belong to such a group are simply *nodes*). These blocks of columns can be treated together as a dense submatrix in the column Cholesky algorithm. A slightly more restrictive definition of supernode is used in the multifrontal algorithm, and has been applied to the column Cholesky algorithm as well [3, 11]. The more restrictive definition requires that the columns in the supernode other than the first have no direct predecessors. This is necessary for the multifrontal algorithm because frontal matrices corresponding to direct predecessors must be assembled into the frontal matrix for a column or supernode prior to the update step. In the column Cholesky algorithm, there is no such requirement. Supernodes can be identified in the elimination tree as chains of consecutively-numbered nodes, where the columns have identical nonzero structures.

There are two points in the computation of a column of L where supernodes can be exploited:

- If any column in a supernode is required for the update to the current column j, then all columns in that supernode are needed. The total contribution of all columns in the supernode can be computed in dense mode and then gathered once into the work array.
- If the current column j is a member of a supernode, then updates from all columns preceding column j in the same supernode can be applied directly to column j. (This is an improvement to the method described in [3] which treats this case the same as the previous one.)

In the parallel factorization, supernodes can be exploited to gain an additional level of task overlap. All columns in a supernode will require updating by the same set of lower-indexed columns not in the supernode (a *sparse update*), as well as by the previous columns within the supernode itself (a *dense update*). The sparse update can be performed in parallel on all columns of the supernode before the dense update. This provides a significant improvement to the performance of the parallel factorization algorithm.

4.4 Data Structures

The principal data structure is a sparse, column-major representation of L. Three arrays are used: a double-precision array containing the values of all nonzeros in L, in order by column and within columns by row; an integer array containing the row indices of the nonzeros; and an m-array containing

the index of the start of each column in the nonzero array. The diagonal matrix D is stored in a separate m-array. This is a static structure that can be constructed once at the start of the primal-dual algorithm, using a *symbolic factorization* procedure [4, 8]. The lower triangle of $A\Theta A^T$ can be stored in the same data structure, and the factorization done in place. Since the contents of Θ are altered at each iteration of the primal-dual algorithm, $A\Theta A^T$ must be re-computed (see below).

An additional data structure is required for column Cholesky, to link the columns that contribute to updating each column, and to locate the nonzero entry in the row corresponding to the column to be updated. For serial (and vector) implementations, [8] describe a set of non-overlapping linked lists, which can be maintained in two m-arrays. There is a list for each row of L. At the start of the algorithm, each column appears on the list corresponding to the row of the first subdiagonal nonzero in the column. As each column k on the list for row j is used to update column j, it is moved from the jth list to the list corresponding to the row of the next nonzero in column k. Since each column appears on only one list at any given time, the lists themselves may be stored in a single m-array. The list headers are contained in the same array, since any column for which the list is non-empty is not yet computed, and hence is not on any list. Finally, for each column k on the list for column j, the index of l_{jk} in the array of nonzeros is stored in a second m-array. This allows the relevant portion of column k (those entries in row j and below), to be located directly without having to search the entire column. Since all columns in a supernode are treated as a unit, only the first column of each supernode need be kept on these lists.

For our experiments with the parallel column Cholesky, we use a static structure, where linked lists corresponding to each row are constructed during preprocessing. This structure requires two arrays with the same number of entries as L has nonzeros. In the future, this structure will be replaced with a dynamic structure, in which the linked lists are updated dynamically in parallel.

One additional m-array is needed to identify the supernodes. For each singleton column j (not in any supernode) the corresponding entry in this array contains j. If j is the first column in a supernode, the jth entry in the array contains the index of the last column in the supernode. Each other column in the supernode whose first column is j also contains j. Thus columns in a supernode can readily be identified and the first column in the supernode can be located directly.

4.5 Construction of $A\Theta A^T$

The value of an element of $M = A\Theta A^T$ is given by

$$m_{ij} = \sum_{k=1}^{n} a_{ik} a_{jk} \theta_{kk}.$$

Again, the ordering of the indices i, j and k determine different algorithms for constructing M. The ijk algorithm computes M one element at a time. The kij algorithm computes M as the sum of scaled outer products of columns of A. Previous implementations of interior point algorithms (notably [1]) use one of these techniques. Both of these methods have drawbacks, however. The ijk algorithm requires two multiplications in its innermost loop. The straightforward kij algorithm is unsuitable for sparse implementation because it is not possible to compute the location in the array of nonzeros into which to accumulate each term. One solution to both of these problems is to save a list of the *elementary products* (products of the pairs $a_{ik}a_{jk}$) and the corresponding locations in the nonzero array. This allows the algorithms to run at full speed, but requires a very large amount of memory for the arrays. In fact, in virtual memory systems the paging activity associated with the construction of $A\Theta A^T$ can slow the algorithm down.

The OB1 implementation avoids both elementary products and the double multiplication in the inner loop, by employing the jki form of the construction algorithm. This method can be interpreted as constructing $A\Theta A^T$ a column at a time. The cost of implementing this method is that A must be accessible in both row-major and column-major order. A is stored in a sparse, column-major data structure, similar to that described above for L. An additional set of three arrays links the entries in each row together on a list (this requires one m-array and an array with an entry for each nonzero) and records the column index of each nonzero entry. It is straightforward to find the entries in column k below row j. These entries can then be scaled by $a_{jk}\theta_{kk}$ and accumulated into the appropriate positions in column j of the L-structure, using a sparse SAXPY. This method has proven to be nearly as fast as the others, and provides a significant savings in memory.

The jki form of the construction step is also readily implemented as a parallel algorithm, since construction of each column is independent of the others.

4.6 Vectorization

The sparse and dense loops that perform a column update contain an apparent *dependence relation* that prevents a vector compiler from automatically vectorizing the loop. In the sparse case the dependence is based on indirect indexing into the work array through the array of row indices. Since the row indices of the nonzeros in any column are distinct, the loop runs correctly when vectorized, but the compiler decision must be overridden explicitly. Similarly, the loop that packs the work array back into the sparse data structure must be explicitly vectorized. Other sparse SAXPY operations appear throughout the code: during the forward and backward triangular system solution, the construction of $A\Theta A^T$, and operations involving A alone, such as the ratio test, and the construction of the right hand side of $A\Theta A^T x = r$. The dense update of a column by other columns in the same supernode also appears to contain a dependency, since the compiler cannot guarantee that the sections of the array of nonzeros corresponding to distinct columns do not overlap. Here again, the compiler incorrectly fails to vectorize the loop and must be overridden manually. In the code tested here, all false dependencies discovered by the VS FORTRAN compiler have been explicitly overridden. There may be additional loops that are currently unanalyzable, but that could be restructured to allow vectorization, for example, by moving print statements for debugging to separate loops.

4.7 Parallel Implementation

The parallel factorization algorithm utilizes self-scheduling processes dispatched at each iteration on all available processors. Coordination of tasks is managed using three lists. Each list is updated inside a critical section, under control of a lock, to prevent simultaneous writes to the same location.

- A list of tasks ready to execute. A "task" represents either a supernode whose predecessors have been computed, or an individual column ready for its sparse update. The action taken in the former case is to replace the task on the list with the tasks corresponding to the individual columns in the supernode (*splitting*). To minimize contention for the lock, all other operations are integrated into the processing of these two types of tasks, as described below.

- A count of the uncompleted predecessors of each column (children of the corresponding node in the elimination tree). As each column j

is computed, the counter corresponding to *Parent*(j) is decremented. When the predecessor count goes to zero, the column is itself ready to compute. Note that, when the first column in a supernode is ready, all columns in that supernode are ready as well.

- A counter of the number of columns in each supernode. As the sparse update for each column in the supernode is completed, this counter is decremented. When its value reaches zero, the supernode is ready for its dense update.

Each processor executes the following algorithm:

1. Get the next task (column j) from the task list.
2. If task j represents a supernode ready to split, schedule the sparse updates for each column and go to step 1. If j represents a single column, go to step 3.
3. Task j represents a sparse update. Perform the update and decrement the column counter for the supernode containing j. If this is the last sparse update in this supernode, then perform the dense update on the supernode and and decrement the predecessor counter for column *Parent*(j). If this counter goes to zero, set $j :=$ *Parent*(j) and go to step 2, else go to step 1.

When the task list is temporarily empty, the current version of the algorithm uses a spin-lock technique to wait for additional work.

5 Computational Experience

We solved several sample problems, some from the Netlib test set [6], and some from other sources, representing various applications of linear programming from stochastic modeling in forestry to shipping, refinery operation and airline crew scheduling. The dimensions of the original problem (the A matrix) are given in Table 1. These figures are after reduction by a preprocessor that fixes variables and removes redundant constraints before performing the minimum-degree ordering. Table 2 gives dimensions of the AA^T matrix and the number of fill-in entries in the L-factor. (Note that the number of nonzeros given for AA^T is for the lower triangle only.) Table 3 gives the percentage of columns of L that are members of a supernode. This

Table 1: Model dimensions

Models	A			
	Rows	Columns	Nonzeros	Density (%)
KPEAR	120	308	631	1.71
BRANDY	126	249	2084	6.64
SHIP04L	317	2118	6101	0.91
TEST3	364	1212	7481	1.70
GROW22	440	946	8252	1.98
SHIP08L	520	4283	11614	0.52
AA2	531	5198	36359	1.32
MILT	586	1338	6642	0.85
NESM	646	2923	13256	0.70
SHIP12L	687	5427	14913	0.40
SCFXM3	846	1371	7558	0.65
CZPROB	927	3523	10669	0.33
GANGES	1137	1681	6740	0.35
80BAU3B	2021	9799	20648	0.10
STOCFOR2	2141	2031	8319	0.19
PIMS2	2405	3120	19141	0.26

is one indicator of the advantage to be expected from exploiting the supernode structure to avoid sparse vector operations. Table 3 also gives the size of the largest supernode in L.

The code used for the vector comparisons was compiled using the IBM FORTRAN version 2.3 compiler under VM/XA and CMS 5.5. The machine was an IBM 3090-600E with the Vector Facility and 256 megabytes of memory. The parallel tests were performed using the IBM Parallel FORTRAN compiler, and run after the machine was upgraded to an IBM 3090-600J.

5.1 Vectorization

Table 4 compares vector and scalar versions of the factorization. For each problem, virtual and total CPU seconds (not including input/output times but including overhead for setting up the L data structure) are given for each of four runs: a standard column Cholesky algorithm (KCHC) and a column Cholesky with supernodes (KCHCS), with the code compiled in

Table 2: AA^T and Cholesky factor L

Models	AA^T			L	
	Columns	Nonzeros	Density (%)	Fill-ins	Density
KPEAR	120	247	3.43	666	0.09
BRANDY	126	1981	24.96	2593	0.33
SHIP04L	317	3740	7.44	3977	0.08
TEST3	364	7395	11.10	5015	18.78
GROW22	440	4600	4.76	4018	8.92
SHIP08L	520	6110	4.52	6442	0.05
AA2	531	27191	19.32	103898	93.16
MILT	586	15308	8.93	10683	15.12
NESM	646	4057	1.95	16976	10.10
SHIP12L	687	8145	3.46	471	3.65
SCFXM3	846	8236	2.30	4736	3.63
CZPROB	927	6616	1.54	388	1.63
GANGES	1137	7484	1.16	29284	0.05
80BAU3B	2021	9533	0.47	39972	0.02
STOCFOR2	2141	12666	0.55	13772	1.15
PIMS2	2405	20201	0.70	43564	2.21

Table 3: Supernodes

Models	% of columns in supernode	Max size of a supernode
TEST3	67.31	51
GROW22	6.36	28
AA2	95.48	143
MILT	88.06	53
NESM	50.16	29
SHIP12L	31.88	15
SCFXM3	62.77	17
CZPROB	4.21	3
STOCFOR2	35.22	27
PIMS2	59.46	120

Table 4: Time for each method in serial and vector mode

Models	Class	Serial test		Vector test		Iter.
		KCHC	KCHC1	KCHC	KCHC1	
TEST3	V	23.74	21.72	8.93	8.13	39
	T	23.88	21.84	9.01	8.21	
GROW22	V	16.01	16.00	8.08	8.12	47
	T	16.15	16.09	8.12	8.19	
AA2	V	321.51	271.88	60.07	50.32	25
	T	323.52	273.41	60.55	50.73	
MILT	V	60.94	61.41	19.85	18.69	50/53
	T	61.31	61.75	20.01	18.81	
NESM	V	87.37	80.48	34.42	31.66	81
	T	87.84	80.93	34.70	31.87	
SHIP12L	V	16.39	16.20	12.11	12.04	38
	T	16.47	16.29	12.18	12.12	
SCFXM3	V	23.67	22.73	13.15	12.67	55
	T	23.80	23.04	13.26	12.76	
CZPROB	V	15.70	15.85	12.42	12.53	46
	T	15.81	15.97	12.53	12.64	
STOCFOR2	V	43.50	41.62	25.96	25.29	41
	T	43.72	41.84	26.09	25.48	
PIMS2	V	203.35	173.17	62.41	53.24	38
	T	204.55	174.14	63.24	53.47	

serial mode (compiled with the NOVECTOR option) and in vector mode (with the VECTOR option and all compiler directives activated). Finally, the number of iterations required by the primal-dual algorithm, including a "pre-factorization" to detect any linearly dependent rows of AA^T, is given. Where two iteration counts are given, the first refers to KCHC and the second to KCHCS. In these cases, the difference in the order of computation causes the termination criterion to be satisfied after different numbers of iterations in each case.

The benefits of careful vectorization are immediately apparent. Savings range from 20% for CZPROB to 80% for AA2. These results indicate that care is required when porting serial code to vector machines; automatic

vectorizing compilers are not a complete solution to the "dusty deck" problem. Vectorization is less advantageous when L is relatively sparse, and more advantageous when L contains columns with many nonzeros. A large fraction of columns in supernodes appears also to increase the benefits of vectorization.

The advantages of supernodes are, of course, most pronounced when there are many columns in supernodes and when the supernodes are large. In CZPROB, for example, the supernode structure is disadvantageous, but in AA2, PIMS2 and NESM the advantage from exploiting supernodes is pronounced. This suggests that a more careful examination of the supernode structure of a problem would be beneficial. One possible step would be to merge columns adjacent to supernodes into the supernodes to increase their size. This would entail including some explicit zero entries in L, but the tradeoff for increasing the portion of dense updates should be worthwhile, up to a point.

5.2 Parallel Implementation

Table 5 describes the results of a standalone test comparing wall-clock times on one, three and six processors. Each test compares a straight parallel column Cholesky (essentially the above algorithm with every column treated as an independent supernode) with the supernode algorithm described above. The scheduling overhead for these algorithms on a single processor, as compared to a uniprocessor implementation, is negligible. (Differences in iteration counts between Tables 4 and 5 are due to adjustments in parameters of the algorithm between the tests.)

Along with the parallel Cholesky algorithm, the following steps were implemented in parallel: construction of $A\Theta A^T$, and computation of Ax and yA.

The performance of the supernode algorithm improves on the standard algorithm by an average of 5% on a single processor, but increases to a 25% average improvement on three processors and a 35% average improvement on six processors. This points to the effectiveness of the parallel sparse update of columns in supernodes. The average speedup of the supernode algorithm when multiple processors are used is 1.60 for three processors and 1.85 for six processors. The maximum speedup achieved was nearly 2.0 on three processors, and nearly 2.4 on six processors (both for the problem MILT).

Due to the hierarchical nature of the tasks and the fact that parallelism

Table 5: Wall Clock Times (seconds)

Problem	iter	P = 1		P = 3		P = 6	
		C	S	C	S	C	S
KPEAR	19	0.42	0.45	0.34	0.40	0.36	0.48
BRANDY	27	1.10	1.12	0.85	0.90	0.82	0.83
SHIP04L	22	2.68	2.71	1.84	1.86	2.13	2.08
TEST3	39	7.41	7.00	5.82	4.49	5.51	4.04
GROW22	30	4.46	4.61	2.61	2.83	2.36	2.57
SHIP08L	24	4.92	4.97	3.39	3.35	3.03	3.04
MILT	49	15.37	14.04	10.64	7.23	9.94	5.90
NESM	66	22.82	21.41	16.78	12.81	15.79	10.95
SHIP12L	27	7.35	7.41	4.64	4.80	4.09	4.33
SCFXM3	39	7.83	7.90	5.10	4.65	4.63	4.19
CZPROB	57	10.44	10.58	8.31	7.71	7.67	7.07
GANGES	34	12.25	11.01	8.96	6.75	8.38	5.51
80BAU3B	77	64.48	61.07	51.30	40.11	47.87	35.16
STOCFOR2	60	21.89	22.23	12.49	12.66	10.19	10.77
PIMS2	32	36.22	32.49	30.02	19.06	28.23	16.24

is exploited only in the factorization step, it is probably not reasonable to expect linear speedup. For larger problems, it is likely that the speedup on six processors would also improve, as the ratio of work in the factorization step would be maintained at a higher level.

6 Conclusions

Our investigations have shown that dramatic improvements in performance of interior point algorithms are possible if they are implemented correctly on parallel and vector computers. We have achieved substantial performance improvements on the IBM 3090 by exploiting its parallel and vector processing capabilities, and the special structure inherent in the problem to be solved.

There are several areas for future research. Better measurement and larger test problems would give better insight into how effectively parallel processors could be used. Additional opportunities for parallelism exist, particularly in the solution of systems involving the factored matrix, as well as other computations. Different methods for factorization, such as the multifrontal algorithm or iterative methods may yet prove more suitable. In addition, problems with special structure may prove particularly well-suited for parallel solution.

The study of interior point methods is still in its infancy. Many issues remain to be resolved before we can say that we have fully exploited the capabilities of advanced computer architectures for this problem. In addition, developments in interior point methods have spurred new research into implementation of the supposedly mature simplex method. Research into these methods and their application in other optimization problems will certainly continue to advance the state of the art in both supercomputing and optimization well into the future.

Acknowledgements

This work was supported in part by the Center for Research on Parallel Computation through NSF Cooperative Agreement No. CCR-8809615.

Computational testing was conducted using the Cornell National Supercomputer Facility, a resource of the Center for Theory and Simulation in Science and Engineering (Cornell Theory Center), which receives major funding from the National Science Foundation and IBM Corporation,

with additional support from New York State and members of the Corporate Research Institute. The author would like to thank the Theory Center consulting staff for their invaluable assistance.

Thanks are also due to Ho-Won Jung for assistance with early coding and testing.

References

[1] I. Adler, N. Karmarkar, M. G. C. Resende, and G. Veiga. Data structures and programming techniques for the implementation of Karmarkar's algorithm. *ORSA Journal on Computing*, 1(2):84–106, 1989.

[2] I. Adler, M. G. C. Resende, G. Veiga, and N. Karmarkar. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming*, 44(3):297–336, 1989.

[3] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *International Journal of Supercomputing Applications*, 1(4):10–30, 1987.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York NY, 1986.

[5] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric sets of linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[6] D. M. Gay. Electronic mail distribution of linear programming test problems. *COAL Newsletter*, 13:10–13, December 1985.

[7] A. George, M. T. Heath, and J. W.-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its Applications*, 7:165–187, 1986.

[8] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliffs NJ, 1981.

[9] J. G. Lewis and H. D. Simon. The impact of hardware gather/scatter on sparse Gaussian elimination. *SIAM Journal of Scientific and Statistical Computing*, 9(2):304–311, March 1988.

[10] J. W.-H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, June 1985.

[11] J. W.-H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1987.

[12] J. W.-H. Liu. The role of elimination trees in sparse factorization. Technical report, Dept. of Computer Science, York University, North York, Ontario, Canada, 1988.

[13] I. J. Lustig, R. E. Marsten, and D. F. Shanno. Computational experience with a primal-dual interior point method for linear programming. Technical Report J-89-11, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta GA, 1989.

[14] R. E. Marsten, M. J. Saltzman, D. F. Shanno, G. S. Pierce, and J. F. Ballintijn. Implementation of a dual affine interior point algorithm for linear programming. *ORSA Journal on Computing*, 1(4):287–297, 1989.

[15] K. A. McShane, C. L. Monma, and D. F. Shanno. An implementation of a primal-dual interior point method for linear programming. *ORSA Journal on Computing*, 1(3):70–83, 1989.

[16] M. J. Saltzman, R. Subramanian, and R. E. Marsten. Implementing an interior point LP algorithm on a supercomputer. In R. Sharda, B. L. Golden, E. Wasil, O. Balci, and W. Stewart, editors, *Impacts of Recent Computer Advances on Operations Research*, pages 158–168. North-Holland, New York NY, 1989.